hakin9

# Construction d'un désassembleur de taille orienté Hooking

Rubén Santamarta

Article publié dans le numéro 4/2006 du magazine *hakin9*



Magazine *hakin9*, Software – Wydawnictwo, ul. Piaskowa 3, 01-067 Varsovie, Pologne, *fr@hakin9.org*

Pratique

# Construction d'un désassembleur de taille orienté Hooking

Rubén Santamarta

Degré de difficulté

**Jour après jour, les chercheurs des programmes malveillants, les analystes juridiques ou les administrateurs doivent faire face aux dangers menaçant la sécurité des systèmes informatiques. Leur objectif est d'expliquer les intrusions non autorisées, de protéger les utilisateurs contre les virus ou de protéger les systèmes contre différents dangers.**

Pour atteindre ces buts, une analyse très détaillée du fonctionnement des logiciels malveillants est nécessaire ; c'est l'ingénierie inverse qui entre ici en jeu.Les concepteurs des programmes malveillants (virus, troyens, rootkits) essaient de rendre cette analyse extrêmement difficile, par exemple à l'aide des techniques empêchant le déboguage, des techniques polymorphiques, des techniques stealth ou des programmes pour la compression des fichiers ; ces derniers non seulement réduisent la taille des fichiers exécutables, mais aussi ajoute une couche protectrice plus ou moins complexe.

Dans ces situations, c'est le temps qui compte avant tout : grâce à une analyse longue et précise, tôt ou tard, nous atteindrons notre but et pourrons connaître tous les détails du danger. Hélas, il arrive que nous n'avons pas assez de temps et dans ce cas il faut optimiser les opérations liées aux procédures de l'analyse. Imaginez un ver exploitant une erreur inconnue d'un programme qui lui permet de se répandre via Internet. Le temps investi dans l'analyse et la compréhension du fonctionnement de ce ver marque la limite entre la vrai catastrophe des utilisateurs et le danger réduit et neutralisé.

De cela, nous prendre des mesures suffisantes pour résoudre chaque type de problèmes auxquels nous pouvons nous heurter.

## Hooking

Comme on peut remarquer, il existe beaucoup d'astuces qui ont pour but de rendre difficile l'utilisation du débogueur (aussi bien au niveau Ring0 que Ring3) qui est un principal outil dans l'ingénierie inverse. De cela, il est nécessaire

### Cet article explique...

- Comment utiliser hooking pour l'analyse des programmes malveillants (malware).
- Comment exploiter Structure Exception Handling pour la création d'un désassembleur de taille.

### Ce qu'il faut savoir...

- Connaître Assembleur x86 y C.
- Connaître Win32 Api et Structure Exception Handling.
- Avoir les notions de base des techniques utilisées par les logiciels malveillants et les virus.

## Techniques utilisées contre désassembleurs et débogueurs

Pendant des années, les concepteurs des programmes malveillants, les auteurs des virus ou même les développeurs des programmes commerciaux, dotaient leur programmes des méthodes anti-debug et anti-disasm. La plupart d'elles sont destinées à détecter si un programme donné est suivi par un débogueur. Si c'est le cas, le programme peut entreprendre différentes actions, en commençant par l'arrêt immédiat du démarrage et en terminant par le redémarrage de l'ordinateur ou même par des opérations plus agressive, ce qui, heureusement, est peu populaire.

- Une très vielle astuce utilisée pour détecter la présence de SoftIce, le débogueur Ring0 le plus connu, employé dans le monde entier dans l'ingénierie inverse, était une tentative d'accéder aux mécanismes créés par l'un de ses pilotes - NtIce.
- L'instruction `RDTSC` dans l'assembleur x86 : un mnémonique de *Read Time−Stamp Counter*. Cette instruction stocke dans `EDX:EAX` (64 bits) la valeur *timestamp* du processeur. Imaginons que `RDTSC` est lancée au début du bloc du code, et la valeur retournée est stockée. À la fin de ce bloc du code, nous lançons encore une fois `RDTSC` et soustrayons la valeur obtenue de la valeur stockée. Quand le programme est démarré de façon ordinaire, le résultat de cette opération aura les valeurs rationnelles, en tenant bien sûr compte de la vitesse et de l'utilisation du processeur. Mais si nous nettoyons ce bloc du code, l'incrément de *timestamp* entre deux lecture sera très élevé par rapport à celui trouvé dans le débogueur.
- La manipulation des interruptions pour modifier le flux du code. Une possibilité très puissante de l'architecture Win32 est la fonction Structure Exception Handling (SEH) qui permet de déterminer les schémas de la fonction *callback* pour contrôler les exceptions. Vu que les débogueurs s'occupent de toutes les exceptions ayant lieu lors du fonctionnement du programme, les procédures définies par le programmeur pour la gestion des exceptions ne seront jamais utilisées. Admettons que nous avons basé le flux de notre programme sur la procédure de ce type. Si après une génération expresse de l'exception (par exemple, au moyen de `xor eax,eax`, et ensuite `mov [eax],eax` ), nous ne parvenons pas à l'espace du code admis, nous sommes probablement sous la surveillance d'un débogueur.
- Autres astuces, moins élaborées, basés sur les propriétés spécifiques de chaque débogueur : les tentatives de retrouver certains types ou titres de fenêtres enregistrés par le programme ou bien des clés dans le registre de *Windows* qui pourraient le dévoiler.

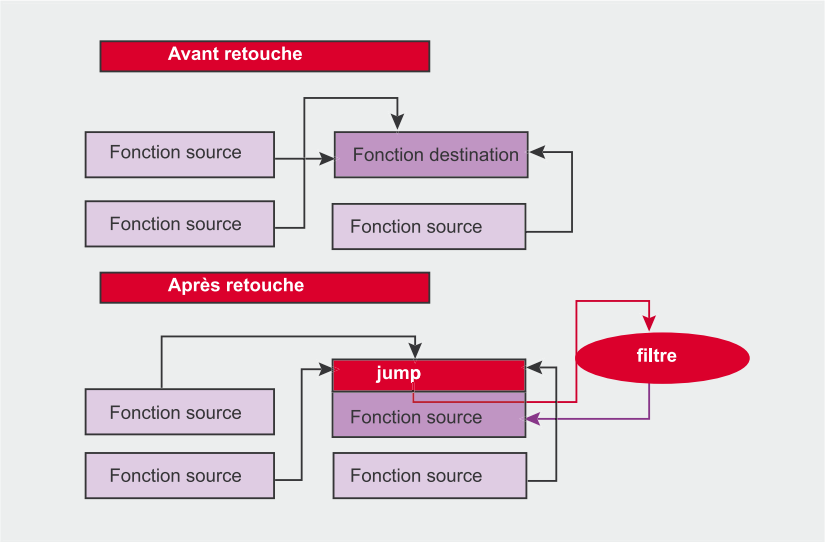


**Figure 1.** *Le schéma de base d'Inline Hooking*

d'élaborer une méthode qui, dans les conditions appropriées, permettrait d'influencer le comportement du fichier exécutable analysé et le modifier.

L'une des techniques les plus utilisées servant à atteindre ce but est hooking.

On peut distinguer différentes techniques d'accrochage en fonction de l'endroit où il a lieu. Chaque type est destiné à d'autres applications. Ainsi, on obtient les types suivants :

- Inline Hooking,
- Import Address Table hooking,
- System Service Table hooking (Ring0),
- Interrupt Descriptor Table hooking (Ring0),
- IRP hooking (Ring0),
- Filter drivers (NDIS,IFS...Ring0).

Dans notre cas, nous allons utiliser la méthode Inline Hooking. Nous l'employons parce qu'elle permet Nous utilisons cette technique parce qu'elle permet de retoucher la fonction à intercepter quand elle est chargée dans la mémoire. Ainsi, nous ne devons pas nous soucier de quel endroit proviennent les appels ou combien de fois ils ont eu lieu, mais nous attaquons directement le noyau. Un appel quelconque de la fonction sera intercepté par notre crochet.

### Interception et modification du flux de code

Admettons que nous voulons intercepter tous les appels API *CloseHandle* qui se produisent lors du démarrage du programme. L'API se trouve dans *kernel32.dll*, donnons un coup d'œil sur ses premières instructions :

```
01 8BFF  mov  edi,edi
02 55  push  ebp
03 8BEC  mov  ebp,esp
04 64A118000000  mov  eax,fs:[00000018]
05 8B4830  mov  ecx,[eax][30]
06 8B4508  mov eax,[ebp][08]
```

Ce bloc du code présente les premiers octets du point d'entrée *CloseHandle*, ce qui signifie que chaque

**Tableau 1.** *Les données disponibles pour la gestion des exceptions, si celle-ci est active*

| T | Données |
|---|---|
| ESP+4 | Pointeur à la structure EXCEPTION_RECORD |
| ESP+8 | Pointeur à la structure ERR |
| ESP+C | Pointeur à la structure CONTEXT_RECORD |

appel de la fonction démarrera sans doute le code précédent. En nous servant du schéma d'*Inline Hooking*, nous remplacerons ces premières instructions par notre propre crochet qui modifiera le flux normal de la fonction dans la direction du filtre.

## Différentes possibilités pour le même objectif

Il existe différentes méthodes de changement du flux dans la direction du notre code. La plus simple consiste à changer les premiers octets de CloseHandle par un saut inconditionnel.

```
01 E9732FADDE   jmp   0DEADBEEF
02 64A118000000   mov   eax,fs:[00000018]
```

Nous avons remplacé 5 premiers octets par un saut vers l'adresse 0xDEADBEEF. Bien sûr, cette adresse n'est pas valide car nous travaillons en mode utilisateur. À cette adresse, il y aurait le code que nous avons introduit dans l'espace adressable du fichier exécutable.

Vu que c'est la méthode la plus simple, elle est aussi la plus facile à détecter parce qu'il est suspect si le point d'entrée d'une fonction système contient le saut inconditionnel vers une autre adresse dans la mémoire. Nous pouvons utiliser une autre option étant la combinaison de `PUSH` + `RET` .

```
01   68EFBEADDE   push   0DEADBEEF
02   C3   retn
03   A118000000   mov   eax,[00000018]
```

Dans ce cas, nous remplaçons les 6 premiers octets. Si nous regardons ce fragment avec attention, nous constatons que le code original *CloseHandle* est effectivement changé, mais non du point de vue des instructions ajoutées, mais aussi certaines d'entre elles sont perdues et d'autres ont changé. C'est une question à discuter. Bien que nous avions atteint notre objectif et intercepté tous les appels de la fonction, la modification étaient si importante qu'elle a entraîné les anomalies du comportement, ce qui en résultat pourrait provoquer la fin inattendue du programme pendant le premier appel *CloseHandle*.

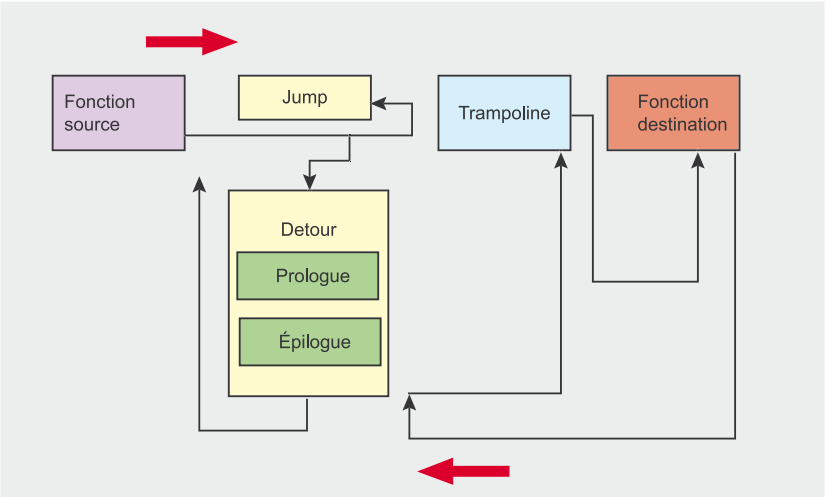


**Figure 2.** *Le schéma de base de la technique Detour*

C'est pourquoi, il faut développer une technique le moins agressive envers le code source qui permettrait à la fonction *accrochée* de fonctionner normalement tout en restant constamment sous contrôle. Cette technique est appelée Detour (elle a été présenté par Galen Hunt et Doug Brubacher des laboratoires Microsoft)

## Detour

Par rapport à Inline hooking, la technique Detour implémente deux nouvelles conceptions : la fonction Detour et la fonction Trampoline.

- *Fonction Detour* doit se composer d'une partie dans laquelle sont réalisées les premières opérations sur les données obtenues, ensuite vient l'appel à la fonction Trampoline et à la fin, se trouve la partie du code qui sera lancé après la terminaison de la fonction Trampoline.
- *Fonction Trampoline* contient aussi bien les instructions de la fonction cible, remplacées complètement par le saut inconditionnel, que celles qui ont été changées partiellement. Ensuite, il y a un saut à l'instruction suivante correspondant à la fonction cible.

Ainsi, nous pouvons résoudre le problème lié aux données perdues ou aux instructions modifiées qui se produisait dans Inline Hooking. Il s'agit de sauver ces instructions dans la fonction Trampoline pour qu'elles puissent être exécutées. Ensuite, nous sautons à une instruction suivante où la fonction cible fonctionnera normalement. Une fois la fonction cible terminée, nous reprenons le contrôle sur la fin de la fonction Detour. Elle est capable de récupérer le chemin de démarrage en restaurant le contrôle de la fonction source ou la possibilité de réaliser un autre type d'opération.

Mais comment savoir combien d'instructions faut-il copier dans la fonction Trampoline à partir de la fonction cible ? Chaque fonction cible sera différente, et de cela il

### Codes des exceptions

Une exception due à un accès à la zone de la mémoire invalide et une exception produite à la suite de l'opération de division par 0 ne peuvent pas être traitées de la même façon. Le système identifie chaque situation pour faciliter la tâche de la gestion des exceptions. Les codes des exceptions les plus répandues sont :

- C0000005h – Violation des droits d'accès aux opérations de lecture ou d'écriture.
- C0000017h – Pas de mémoire libre.
- C00000FDh – *Stack Overflow.*

Les codes suivants sont les plus importants pour notre projet :

- 80000003h – *Breakpoint* généré par l'instruction `int 3`.
- 80000004h – *Single step* généré par l'activation de *Trap Flag* dans le registre *EFLAGS.*

sera impossible de copier le nombre d'octets déterminé car les instructions pourraient être coupées. Ce problème est résolu à l'aide du désassembleur de taille.

### Désassembleurs de taille

Les désassembleurs de taille diffèrent des désassembleurs ordinaires par leur fonction qui consiste à obtenir la taille des instruction et pas leur présentation. Ce type de désassembleurs était utilisé par les virus *cavités*, polymorphiques, etc. C'est pourquoi, les deux des désassembleurs de taille (les vraie perles de l'optimisation) les plus utilisés ont été programmés par les concepteurs des virus connus : Zombie et RGB.

Ils sont basés sur un désassembalge statique des instructions. Pour cela, ils utilisent les tableaux des codes des opérations de l'architecture sur laquelle ils fonctionnent, dans ce cas x86.

Outre le fait qu'ils sont utilisés pour la création des virus complexes, on l'emploie aussi pour le hooking car ils permettent de résoudre le problème dont nous avons parlé auparavant.

À partir du potentiel de Structure Exception Handling, nous expliquerons la technique innovatrice de la création des désassembleurs dynamiques de taille. Au boulot.

**Tableau 2.** *Champs de la structure EXCEPTION_RECORD*

| Offset | Données |
|---|---|
| + 0 | ExceptionCode |
| + 4 | ExceptionFlag |
| + 8 | NestedExceptionRecord |
| + C | ExceptionAddress |
| + 10 | NumberParameters |
| + 14 | AdditionalData |

## Utilisation de Structure Exception Handling (SEH)

Premièrement, nous devons nous concentrer sur les caractéristiques du problème auquel nous voulons trouver la solution.

- Les premières instructions des fonctions cibles ne diffèrent pas trop, mais les différences sont aussi importantes qu'il est nécessaire d'envisager chaque cas séparément.
- Le saut inconditionnel (jmp) ou Push + ret n'occupent pas plus de 6 octets. Il faudra analyser de 4 à 5 instructions au maximum.
- Les premières instructions exécutent généralement les opérations liées à l'ajustement de la pile.
- L'idée consiste à pouvoir lancer ces premières instructions dans l'environnement contrôlé, ce qui nous permettra de calculer leur longueur.

Pour savoir comment construire cet environnement, il faut saisir les informations apportées par SEH.

Pour chaque exception que a eu lieu dans le code protégé par la fonction SEH définie pour le thread, la gestion définie dispose des données suivantes.

Si une exception se produit, le système démarre la gestion des exceptions pour qu'elle décide des actions ultérieures. À ce moment `esp` pointe vers différentes structures.

Dans la structure EXCEPTION_RECORD, l'attention est attirée sur les champs *ExceptionCode* et *ExceptionAddress :*

- ExceptionCode identifie le type de l'exception créée. Dans le système, différents codes sont définis pour chaque type ; de plus, il est possible de définir nos propres codes pour personnaliser les exceptions via API RaiseException.
- ExceptionAddress est une adresse de la mémoire appartenant à l'instruction créée par une exception ; elle correspondrait au registre eip au moment où une exception s'est produit.

Une autre structure de base qu'il faut connaître est CONTEXT. Cette structure contiendra toutes les valeurs des registres au moment où l'exception se produit.

Il ne faut pas oublier que c'est la possibilité de contrôler chaque instruction démarrée est la plus importante, comme cela se fait dans le débogueur. En fait, notre désassembleur aura toutes les fonctionnalités du débogueur.

## Programmation du désassembleur de taille

Premièrement, il faut définir l'environnement dans lequel nous démarrerons les fonctions surveillées ; ainsi, nous appellerons les instructions appartenant à la fonction cible dont la longueur nous voudrons connaître, pour pouvoir ensuite les copier entièrement dans la fonction *Trampoline*.

Pour cela, premièrement il faut déterminer l'étendue de SEH, SEH_SEHUK étant une routine gérant les exceptions qui se produisent.

```
01  push  dword SEH_SEHUK
02  push  dword  [fs:0]
03  mov  [fs:0], esp
```

Dès ce moment, tout le code démarré après ces instructions sera protégé. L'étape suivante consiste à copier un nombre déterminé d'octets qui contiendront les *instructions surveillées*, à partir de la *fonction cible* vers la zone réservée à l'intérieur de notre code. Sa taille peut changer. Dans notre cas, nous avons choisi 010h car elle est assez large pour contenir en entier les premières instructions.

```
04..mov  esi,TargetFunction
05  mov..edi,Code
06..push  010h
07  pop  ecx
08..rep  movsb
```

Quand nous sommes arrivés à ce point, il ne nous reste qu'une seule étape avant de démarrer les instructions surveillées. Voyons :

```
09  int 3
10  Code:
11   ModCode  times 12h db (90h)
```

ModCode est la zone dans laquelle nous avons copié les instructions surveillées ; avant d'arriver à ce point, nous avons mis la constante `int 3`. Pourquoi ? Il y en plusieurs raisons :

- Comme nous l'avons déjà mentionné, les premières instructions d'une fonction cible quelconque réalisent d'habitude les opérations de modification de la pile. De cela, il faut s'assurer que notre pile ne sera pas détruite par ces actions. Après l'appel d'`int 3`, une exception qui nous permettra d'accéder à *SEH_SEHUK* (notre gestion), est générée. Ainsi, en accédant à la structure *CONTEXT*, nous sauvons les registres *ESP et EBP* pour restaurer l'état de notre pile aux valeurs précédentes après la terminaison de l'analyse.
- L'activation de *Trap Flag* dans le registre *EFLAGS*. Grâce à cette interruption, après le démarrage de l'instruction suivante, l'exception Single Step sera automatiquement générée. Ainsi, le contrôle est restauré. Nous avons obtenu la même information qu'à la suite des opérations effectuées pas à pas.

Consultons ces points créés dans le code de l'assembleur :

```
12  SEH_SEHUK:
13  mov   esi, [esp + 4]
    ; EXCEPTION_RECORD
14  mov   eax, [esi]
    ; ExceptionCode
15  test   al,  03h
    ; Int3 Exception Code
16  mov   eax, [esi + 0Ch]
    ; Eip Exception
17  mov   esi, [esp + 0Ch]
    ; CONTEXT record
18  mov   edx, [esi + 0C4h]
    ; Esp Exception
19  jz   Int1h
20  mov   eax, [esi + 0B4h]
    ; Ebp Exception
21  mov   [OrigEbp], eax
22  mov   [OrigEsp], edx
2  inc   dword [esi + 0B8h]
    ; Eip++ (Int3->Instruction suivante)
24  mov   eax,Code
25  mov   [PrevEip],eax
26  jmp   RetSEH
```

On voit que dans la ligne 15, nous comparons `al` avec 03 pour savoir si l'exception a été produite par `int 3` (code de l'exception *80000003h)*, ou au contraire, il s'agit d'une exception provoquée par *Trap Flag* ou un autre évènement. S'il s'agit de l'exception *BreakPoint*, nous effectuerons les opérations expliquées auparavant (lignes 20, 21, 22).

Il est nécessaire de modifier *EIP* du contexte pour qu'au moment de la restauration du contrôle dans le système, il pointe vers l'instruction qui vient après `int 3`, autrement, nous serions dans une situations sans issue. Pour éviter cela, comme on voit dans la ligne 23, nous incrémentons la valeur d'une unité. Cela est dû au fait que le code de l'opération pour `int 3` a 1 octet.

**Tableau 3.** *Le champ CONTEXT appartenant aux registres généraux et de contrôle*

| Offset | Registre |
|---|---|
| + 9C | EDI |
| + A0 | ESI |
| + A4 | EBX |
| + A8 | EDX |
| + AC | ECX |
| + B0 | EAX |
| + B4 | EBP |
| + B8 | EIP |
| + BC | CS |
| + C0 | EFLAGS |
| + C4 | ESP |
| + C8 | SS |

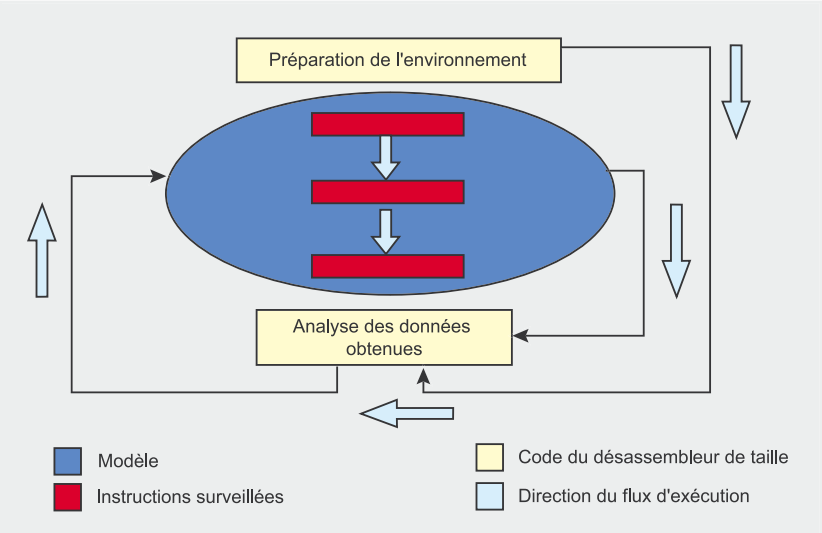


**Figure 3.** *Le schéma du fonctionnement de notre désassembleur de taille*

# OFFRES DE COUPLAGE

12 N°s


+ 6 N°s


~~147 €~~

**99 €**

12 N°s


+ 6 N°s


12 N°s


+ 6 N°s


12 N°s


+ 6 N°s


☐ **OUI, je m'abonne et désire profiter des offres spéciales de couplage**

| | Référence de l'offre : | Prix : | Qté : | À partir du numéro : |
|---|---|---|---|---|
| ☐ | 12 N°s Linux+ DVD + 6 N°s PHP | 99 € | | |
| ☐ | 12 N°s Linux+ DVD + 6 N°s Hakin9 comment se défendre ? | 99 € | | |
| ☐ | 12 N°s Linux+ DVD + 6 N°s SDJ Extra | 99 € | | |
| ☐ | 12 N°s Linux+ DVD + 6 N°s Programmation sous Linux | 99 € | | |

## *Abonnez – vous !!!*

- par courrier en nous renvoyant le bon ci-dessous
- par le web, sur notre site : *http://buyitpress.com/fr*
- par téléphone entre 10-17 au 00 48 22 887 13 44
- par fax au 00 48 22 887 10 11

## *Commande*

Merci de remplir ce bon de commande et de nous le retourner par fax : 0048 22 887 10 10 ou par courrier : Software-Wydawnictwo Sp. z o. o., Piaskowa 3, 01-067 Varsovie, Pologne ; E-mail : *abonnement@software.com.pl*

**Prénom Nom** ..... **Entreprise** .....

**Adresse** .....

**Code postal** ..... **Ville** .....

**Téléphone** ..... **Fax** .....

**E-mail** .....

Je règle par :

☐ **Carte bancaire n° CB** |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_| **expire le** |_|_|_|_| **date et signature obligatoires**

**type de carte** ..... **code CVV/CVC** |_|_|_|

☐ **Virement bancaire :**

**Nom banque : Société Générale Chasse/Rhône**

| banque | guichet | numéro de compte | clé Rib |
|---|---|---|---|
| 30003 | 01353 | 00028010183 | 90 |

**IBAN : FR76 30003 01353 00028010183 90**

**Adresse Swift (Code BIC) : SOGEFRPP**

Dans la ligne 25, nous enregistrons l'adresse de la mémoire où commencent nos *instructions surveillées*, ce qui nous permettra au moment de l'émulation des appels et des sauts conditionnels et inconditionnels.

## Analyse des instructions surveillées

Jusqu'alors, tout ce que nous avons présenté, pourrait être contenu dans le bloc *préparation de l'environnement*. Nous allons commencer à examiner le code appartenant à l'analyse des instructions surveillées.

**Listing 1.** *Observation du code*

```
27 Int1h:
28 mov  ecx, eax
29 sub  eax, [PrevEip]
30 cmp  ax, 10h
31 jb  NoCall
32 mov  ebx, dword [edx]
33 mov  edx,ebx
34 sub  ebx, [PrevEip]
35 cmp  bl, 7
36 jbe  HabemusCall
37 mov  edi,[PrevEip]
38 inc  edi
39 inc  edi
40 mov  dword [esi + 0B8h], edi
41 mov  ecx,edi
42 jmp  NoCall
43 HabemusCall:
44 mov  dword [esi + 0B8h], edx
45 mov  ecx,edx
46 NoCall:
47 mov  [PrevEip], ecx
48 sub  ecx, Code
49 cmp  ecx, [HookLength]
50 jge  Success
51 RetSEH:
52 or  word [esi + 0C0h], 0100h
     ; Activation de Trap Flag
53 xor  eax,eax
54 ret
55 Success:
56 mov  [LenDasm], ecx
   ;Nous restaurons la longueur
de l'instruction
57 mov   esp, [OrigEsp]
     ;Nous récupérons Esp
58 mov   ebp, [OrigEbp]
     ; Nous récupérons  Ebp
59 pop  dword [fs:0]
   ;Nous nettoyons l'étendue SEH
60 add  esp, 4
     ;Nous ajustons la pile
```

### Objectif

Le but de cette partie du code est l'analyse de la longueur des *instructions surveillées* jusqu'à trouver la valeur supérieure ou égale à celle qui occuperait notre crochet, indépendamment du fait, si ce serait un saut inconditionnel (`jmp` 5 octets) ou `push+ ret` (6 octets). Imaginons par exemple que le type de notre crochet est `push+ret` et nous essayons d'accrocher *CloseHandle*. Nous indiquons au désassembleur que notre crochet occupe 6 octets (HookLength = 6). Il commencera alors à compter la longueur de la première instruction

```
01 8BFF  mov  edi,edi
```

Taille 2 octets. Vu qu'il est inférieure à 6, il continue comme suit

```
02 55  push ebp
```

Taille 1 octet +2 octets de l'instruction précédente = 3 octets. Toujours inférieur à 6.

```
03 8BEC  mov ebp,esp
```

Taille 2 octets +3 octets des précédents = 5 octets. On continue

```
04 64A118000000  mov eax,fs:[00000018]
```

Taille 6 octets +5 octets des précédents = 11 octets. C'est déjà fait !

Notre désassembleur de taille retournera 11. Que cela signifie-t-il ? Cela veut dire que pour un crochet composé de six octets, pour ne pas perdre aucune instruction, il faut copier onze octets de la *fonction cible* dans la *fonction Trampoline*.

### Analyse des données

Nous avons ici un bloc initial de l'analyse. Jusqu'à la ligne 46, nous avons l'algorithme permettant d'émuler les appels et les sauts. Cet algorithme consiste à vérifier le décalage entre *EIP* dans lequel l'exception s'est produite et la valeur précédente du registre. Si ce décalage est important, nous avons à faire avec un appel ou un saut, c'est pourquoi nous passerons à la récupération du contexte pour qu'il pointe vers une instruction surveillée suivante au lieu d'exécuter le processus jusqu'à l'adresse 4

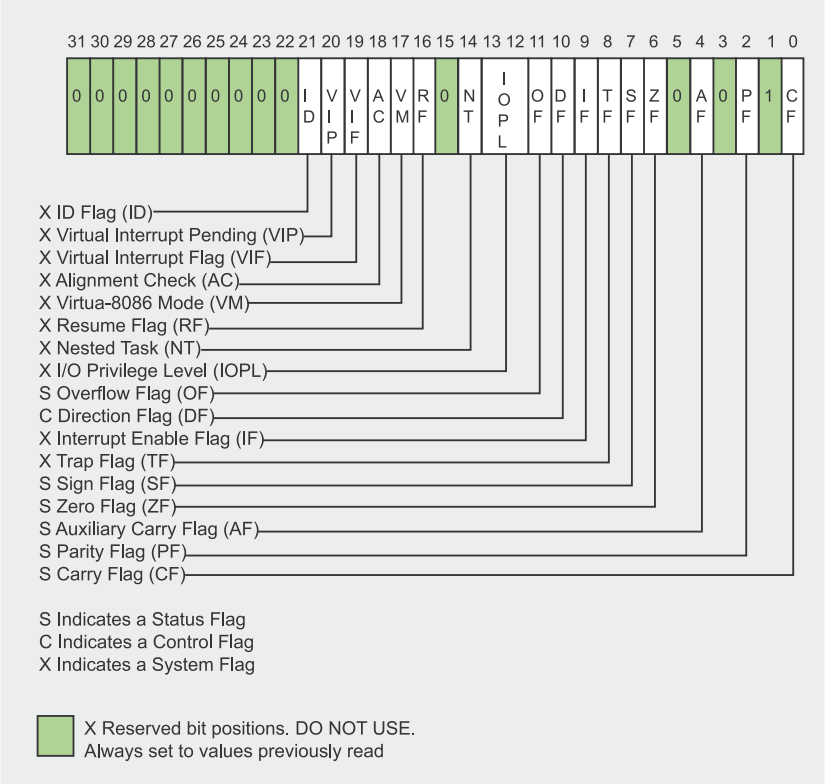


**Figure 4.** *Le registre EFLAGS*

laquelle pointait l'appel ou le saut. Nous compterons aussi avec notre compteur combien d'octets occupe notre instruction.

Dans les lignes 47, 48, 49, nous vérifions s'il y a suffisamment d'instructions pour mettre notre crocher.

La partie la plus principale du désassembleur sont les lignes 52, 53 et 54. Là, nous activons *Trap Flag* en configurant à 1 le bit approprié dans le registre *EFLAGS*. C'est la base pour le nettoyage pas à pas.

À partir de moins de 256 octets, nous avons construit un désassembleur de taille tout à fait opérationnel. Maintenant nous pouvons l'appliquer dans la pratique.

## Application pratique pour l'analyse d'un programme malveillant

On utilise les techniques differentes :

- Pour nos besoins, nous allons créer un programme qui introduira et démarrera du code dans le fichier exécutable qui lui sera passé en paramètre. Les techniques d'insertion du code dans les processus sont bien connues. Il en existe différents types, mais toutes sont basées pratiquement sur les mêmes API.
- *VirtualAllocEx* sert à allouer de la mémoire dans le processus cible. Le code sera introduit dans cet espace mémoire. Pour cela, nous utiliserons *WriteProcessMemory*.
- Au moment du démarrage du code, nous pouvons choisir parmi *CreateRemoteThread* ou *SetThreadContext*.

Mais nous nous servirons d'une autre fonction : *QueueUserAPC*.

### Domaines d'application

Ce programme introduit dans la calculette Windows un petit code qui entraîne l'affichage d'une boîte de message (*Message* Box). La calculette ne s'affichera pas après le démarrage de ce code. Imaginons qu'au lieu d'un code innocent, nous introduisons un code malicieux d'un ver. Imaginons aussi que ce petit code était compacté avec le programme contenant plusieurs types de protections anti-déboguage et anti-désassemblage. Nous devons vite savoir comment il s'infiltre dans un autre fichier exécutable et quel code il introduit. Pour tout cela, il serait nécessaire d'effectuer sur-le-champ le désassemblage du fichier exécutable, mais comme nous l'avions dit, il contient un programme de compactage (packer) qui empêche notre analyse et l'utilisation du débogueur. Il ne reste pas aussi longtemps dans la mémoire qu'il soit possible de le décompresser sur le disque dur à l'aide d'un outil de décompactage (*ProcDump*...). En fait, le fichier exécutable ne reste dans la mémoire que pendant des dizaines de secondes empêchant également de capturer son image. Que pouvons-nous faire ?

Pour résoudre ce problème, on pourrait accrocher *ExitProcess*, la

**Listing 2.** *ACPInject*

```
#include <stdio.h>
#include <windows.h>
typedef BOOL (WINAPI *PQUEUEAPC)(FARPROC,HANDLE,LPDWORD);
int main(int argc, char *argv[])
{
PROCESS_INFORMATION strProces;
STARTUPINFOA                strStartupProces;
PQUEUEAPC                   QueueUserApc;
DWORD                       MessageAddr,Ret1,Ret2,Longueur;
char                        *szExecutableName;
unsigned char               Snippet[]=
"\x90"                     /* nop        */
"\x6A\x00"                 /* push NULL  */
"\x6A\x00"                 /* push NULL  */
"\x6A\x00"                 /* push NULL  */
"\x6A\x00"                 /* push NULL  */
"\xB9\x00\x00\x00\x00"     /* mov ecx,MessageBox*/
"\xFF\xD1" ;               /* Call ecx   */
Longeur = (DWORD) strlen(
"c:\\windows\\system32\\calc.exe") + 1;
ZeroMemory( &strStartupProces, sizeof( strStartupProces) );
strStartupProces.cb = sizeof( strStartupProces );
ZeroMemory( &strProces, sizeof( strProces) );
szExecutableName = (char*) malloc( sizeof(char) * Długość);
if( szExecutableName  )  strncpy(szExecutableName,
"c:\\windows\\system32\\calc.exe",Longueur);
else                          exit(0);
_QueueUserApc = (PQUEUEAPC)GetProcAddress( GetModuleHandle (
"kernel32.dll" ), "QueueUserAPC");
MessageAddr = (DWORD) GetProcAddress ( LoadLibraryA(
"user32.dll") , "MessageBoxA" );
// U32!MessageBoxA
*( DWORD* )( Snippet + 10 ) = MessageAddr;
  Ret1 = CreateProcessA( szExecutableName,   NULL, NULL, NULL,
                    0, CREATE_SUSPENDED,
                    NULL,NULL,
                    &strStartupProces,&strProces);
 Ret2 = (DWORD) VirtualAllocEx( strProces.hProcess,NULL,sizeof(Snippet),
                    MEM_COMMIT,
                    PAGE_EXECUTE_READWRITE);

  WriteProcessMemory(strProces.hProcess,(LPVOID)Ret2,  Snippet,
                      sizeof(Snippet), NULL);
 _QueueUserApc((FARPROC)Ret2,strProces.hThread,NULL);
  ResumeThread(strProces.hThread);
 return 0;
}
```

**Listing 3.** *Outils API*

```
unsigned char HookCode[]=
"\x90"
/* nop                */
"\x68\x38\x03\x00\x00"
/* push 3E8h          */
"\x6A\x6E"
/* push 6Eh           */
"\xB9\x00\x00\x00\x00"
/* mov ecx,00h        */
"\xFF\xD1"
/* Call K32!Beep      */
"\x68\x00\x00\x00\x00"
/* push dword 00h     */
"\xB9\x00\x00\x00\x00"
/* mov ecx,00h        */
"\xFF\xD1"
/* Call K32!Sleep     */
"\x90\x90\x90\x90"
/* Espace pour les instructions
surveillées  */
"\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\x68\x00\x00\x00\x00"
/* push dword 00h     */
"\xC3"
/* ret                */
"\x90";
/* nop    */
unsigned char ExitHook[]=
"\x68\x00\x00\x00\x00"
/* push dword 00h     */
"\xC3";
/* ret                */
```

**Listing 4.** *Nous construisons notre HookCode à l'aide des adresses obtenues*

```
printf("[+]Rebuilding
HookCode...");

// K32!Beep
*( DWORD* )( HookCode + 9 ) =
                    BeepAddr;

// Sleep param
Parameter = atoi( argv[2] );
*( DWORD* )( HookCode + 16 ) =
                    Parameter;

// K32!Sleep
*( DWORD* )( HookCode + 21 ) =
                    SleepAddr;

*( DWORD* )( HookCode + 48) =
HookAddr + LenDasm;

printf("[OK]\n");
```

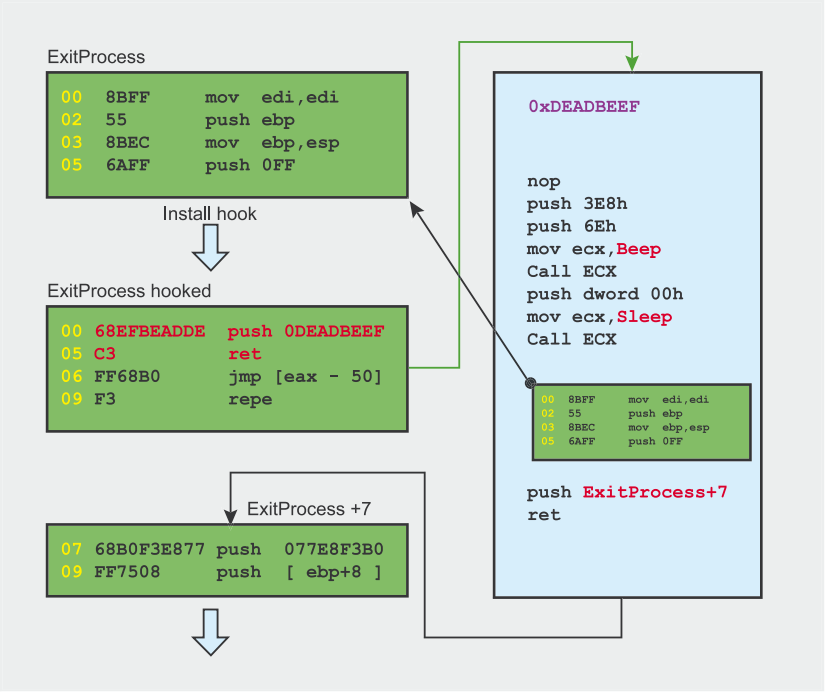


**Figure 5.** *Le schéma d'accrochage d'ExitProcess*

figer dans la mémoire du processus (à l'aide de *Sleep*) et le maintenir dans cet état aussi longtemps qu'il soit possible de la capturer et ensuite reconstruire cette capture binaire pour le désassembler et restaurer à l'état initial. Pourquoi accrocher *ExitProcess* ? 90% de programme malveillant, après avoir arrivé à *ExitProcess* est décompressé dans la mémoire. Il peut arriver que seule une partie du fichier exécutable se décompacte, mais cela n'est pas trop fréquent parce que c'est trop difficile à concevoir. Quant à nous, nous nous concentrerons à construire un outil qui nous permettrait de s'accrocher rapidement à une API d'une dll quelconque utilisée par un programme malveillant. Pour ce faire, nous allons utiliser bien sûr nous nouveau désassembleur de taille.

En tant que technique de hooking, nous utiliserons Inline Hooking avec certains éléments de la technique *Detour*

*HookCode* contient les éléments étant le prologue de la fonction *Detour*. La tâche de ce prologue consiste à nous informer que le programme malveillant a atteint *ExitProcess* par un signal sonore en appelant API *Beep*. Ensuite, comme nous avions mentionné, nous appellerons *Sleep* au moyen du paramètre envoyé via la ligne de commande. Ce paramètre sera assez fort pour nous permettre les opérations de capture, ainsi que d'autres que nous voudrions exécuter. Une fois le prologue terminé, les premières instructions d'*ExitProcess* seront démarrées (les instructions contrôlés par le désassembleur

## Sur Internet

- *http://www.reversemode.com/index.php?option=com_remository&Itemid=2&func=select&id=8* – Le code source complet de toutes les applications mentionnées dans l'article. Le désassembleur de taille. L'exemple d'un programme malveillant et d'une application pour hooking,
- *http://research.microsoft.com/~galenh/dfPublications/HuntUsenixNt99.pdf* – Detours: Binary Interception of Win32 Functions,
- *http://msdn2.microsoft.com/en-us/library/ms253960(VS.80).aspx* – Structure Exception Handling in x86.

**Listing 5.** *ProcessusA*

```
Ret1 = CreateProcessA( szExecutableName, NULL, NULL, NULL,0,
                                    CREATE_SUSPENDED,NULL,NULL,
                                    &strStartupProceso,&strProceso);

 if( !Ret1 ) ShowError();
 printf("[OK]\n");


 printf("[+]Allocating remote memory...");
 Ret2 = (DWORD) VirtualAllocEx( strProceso.hProcess,NULL,sizeof(HookCode),
                                    MEM_COMMIT,
                                    PAGE_EXECUTE_READWRITE);
```

**Listing 6.** *Nous restaurons ExitHook*

```
 *( DWORD* )( ExitHook + 1 ) = Ret2;

 printf("[OK]->Address : 0x%x",Ret2);
 printf("\n[+]Hooking %s...",argv[4]);
 printf("\n\t[-]Reading %d bytes from %s Entry Point ...", LenDasm, argv[4]);
 Ret1 =(DWORD) memcpy( (LPVOID)( HookCode + 27 ),(LPVOID)HookAddr, LenDasm);
 if( !Ret1 ) ShowError();
 printf("[OK]\n");
 printf( "\t[-]Hooking %s...", argv[4] );

 Ret1=0;
 while( !Ret1 )
 {
      ResumeThread(strProceso.hThread);
      Sleep(1);
      SuspendThread(strProceso.hThread);
      Ret1 = WriteProcessMemory(strProceso.hProcess, (LPVOID)HookAddr,
/ * Nous retouchons la fonction cible */
      ExitHook, HookLength, NULL);
/* dans la mémoire */
 }

 printf("[OK]\n");

 printf("\t[-]Injecting Hook...");
 Ret1 = WriteProcessMemory(strProceso.hProcess,(LPVOID)Ret2,
/* Nous copions le code dans l'espace adressable*/
HookCode, sizeof(HookCode), NULL);

/* du processus récemment créé */
/* Nous permettons au processus de se dérouler */
ResumeThread(strProceso.hThread);
```

de taille), et ensuite, le contrôle de l'instruction convenable suivante (*ExitProcess*+7) sera restauré.

Ensuite, il faut reconstruire *HookCode* et *ExitHook* au moyen des adresses de la mémoire API et des valeurs obtenues à partir du désassembleur de taille.

Ensuite, nous créons un processus en mode suspendu et nous allouons de la mémoire dans l'espace adressable.

À la fin, nous restaurons *ExitHook* à l'aide de l'adresse obtenue via *VirtuaAllocEx*, nous retouchons *Entry Point* de la fonction cible (dans ce cas ExitProcess) au moyen d'*ExitHook*.

La façon d'appeler le programme était le suivant :

```
congrio c:\acpinject.exe 10000
 Kernel32.dll ExitProcess
```

- en premier argument, nous avons path malware,
- en deuxième argument, l'intervalle en millisecondes du passage à *Sleep,*
- le troisième argument est DLL qui exporte la fonction cible,
- la *fonction cible* est ici *ExitProcess.*

## Conclusion

Comme on a pu voir dans l'article, les différents domaines de l'ingénierie inverse se convergent dans un point. Nous avons couplé différentes techniques, comme par exemple les désassembleurs de taille, le et l'introduction des processus pour faciliter notre analyse des programmes malveillants.

Paradoxalement, ces mêmes techniques sont exploitées par les programmes malicieux, ce qui aide aussi le développement de l'ingénierie inverse. Plus les rootkits, les virus, etc. sont complexes, plus l'analyse des techniques utilisées et des façons dont elles sont appliquées est détaillée. Ainsi, nous avons une sorte d'une course à laquelle participent les chercheurs, les programmeurs des programmes malveillants ou les auteurs des virus, mais les effets en seront ressentis par millions d'utilisateurs. Pourtant, on ne peut pas nier le fait que les recherches et les innovations proviennent de deux côtés de la barricade. ●


## À propos de l'auteur

L'auteur s'intéressent depuis 16 ans de l'environnement de l'ingénierie inverse et en général de la sécurité informatique. À l'âge de 19 ans, en parfait autodidacte, il a commencé à travailler comme programmeur. Ensuite, il se développé dans les domaines liés au bas niveau, aux programmes anti-virus et à la vulnérabilité aux attaques. Actuellement, il se concentre sur ce dernier domaine.